Learning hashing with affinity-based loss functions using auxiliary coordinates

Ramin Raziperchikolaei Miguel Á. Carreira-Perpiñán
Electrical Engineering and Computer Science, University of California, Merced
http://eecs.ucmerced.edu

January 21, 2015

Abstract

In binary hashing, one wants to learn a function that maps a high-dimensional feature vector to a vector of binary codes, for application to fast image retrieval. This typically results in a difficult optimization problem, nonconvex and nonsmooth, because of the discrete variables involved. Much work has simply relaxed the problem during training, solving a continuous optimization, and truncating the codes a posteriori. This gives reasonable results but is suboptimal. Recent work has applied alternating optimization to the objective over the binary codes and achieved better results, but the hash function was still learned a posteriori, which remains suboptimal. We propose a general framework for learning hash functions using affinity-based loss functions that closes the loop and optimizes jointly over the hash functions and the binary codes. The resulting algorithm can be seen as a corrected, iterated version of the procedure of optimizing first over the codes and then learning the hash function. Compared to this, our optimization is guaranteed to obtain better hash functions while being not much slower, as demonstrated experimentally in various supervised and unsupervised datasets. In addition, the framework facilitates the design of optimization algorithms for arbitrary types of loss and hash functions.

1 Introduction

Information retrieval arises in several applications, most obviously web search. For example, in image retrieval, a user is interested in finding similar images to a query image. Computationally, this essentially involves defining a high-dimensional feature space where each relevant image is represented by a vector, and then finding the closest points (nearest neighbors) to the vector for the query image, according to a suitable distance. For example, one can use features such as SIFT (Lowe, 2004) or GIST (Oliva and Torralba, 2001) and the Euclidean distance for this purpose. Finding nearest neighbors in a dataset of N images (where N can be millions), each a vector of dimension D (typically in the hundreds) is slow, since exact algorithms run essentially in time $\mathcal{O}(ND)$ and space $\mathcal{O}(ND)$ (to store the image dataset). In practice, this is approximated, and a successful way to do this is *binary hashing*. Here, given a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$, the hash function $\mathbf{h}$ maps it to a b-bit vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{-1, +1\}^b$, and the nearest neighbor search is then done in the binary space. This now costs $\mathcal{O}(Nb)$ time and space, which is orders of magnitude faster because typically $b < D$ and, crucially, (1) operations with binary vectors (such as computing Hamming distances) are very fast because of hardware support, and (2) the entire dataset can fit in (fast) memory rather than slow memory or disk.

The disadvantage is that the results are inexact, since the neighbors in the binary space will not be identical to the neighbors in the original space. However, the approximation error can be controlled by using sufficiently many bits and by *learning a good hash function*. This has been the topic of much work in recent years. The general approach consists of defining an objective function that has a small value for good hash functions and minimizing it. Ideally, such an objective function should be minimal when the neighbors of any given image are the same in both original and binary spaces. Practically in information retrieval, this is often evaluated using precision and recall. However, this ideal objective cannot be easily optimized over hash functions, and one uses approximate objectives instead. Many such objectives have been proposed in the literature. We focus here on affinity-based loss functions, which directly try to preserve the original distances in the binary space. Specifically, we consider objective functions of the form

$$\min \mathcal{L}(\mathbf{h}) = \sum_{n,m=1}^{N} L(\mathbf{h}(\mathbf{x}_n), \mathbf{h}(\mathbf{x}_m);\ y_{nm}) \tag{1}$$

where $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$ is the high-dimensional dataset of feature vectors, $\min_{\mathbf{h}}$ means minimizing over the parameters of the hash function $\mathbf{h}$ (e.g. over the weights of a linear SVM), and $L(\cdot)$ is a loss function that compares the codes for two images (often through their Hamming distance $\|\mathbf{h}(\mathbf{x}_n) - \mathbf{h}(\mathbf{x}_m)\|$) with the ground-truth value y_{nm} that measures the affinity in the original space between the two images $\mathbf{x}_n$ and $\mathbf{x}_m$ (distance, similarity or other measure of neighborhood). The sum is often restricted to a subset of image pairs (n, m) (for example, within the k nearest neighbors of each other in the original space), to keep the runtime low. Examples of these objective functions (described below) include models developed for dimensionality reduction, be they spectral such as Laplacian Eigenmaps (Belkin and Niyogi, 2003) and Locally Linear Embedding (Roweis and Saul, 2000), or nonlinear such as the Elastic Embedding (Carreira-Perpiñán, 2010); as well as objective functions designed specifically for binary hashing, such as Supervised Hashing with Kernels (KSH) (Liu et al., 2012), Binary Reconstructive Embeddings (BRE) (Kulis and Darrell, 2009) or Semi-supervised sequential Projection Learning Hashing (SPLH) (Wang et al., 2012).

If the hash function $\mathbf{h}$ was a continuous function of its input $\mathbf{x}$ and its parameters, one could simply apply the chain rule to compute derivatives over the parameters of $\mathbf{h}$ of the objective function (1) and then apply a nonlinear optimization method such as gradient descent. This would be guaranteed to converge to an optimum under mild conditions (for example, Wolfe conditions on the line search), which would be global if the objective is convex and local otherwise (Nocedal and Wright, 2006). Hence, optimally learning the function $\mathbf{h}$ would be in principle doable (up to local optima), although still slow because the objective can be quite nonlinear and involve many terms.

In binary hashing, the optimization is much more difficult, because in addition to the previous issues, the hash function must output binary values, hence the problem is not just generally nonconvex, but also nonsmooth. In view of this, much work has sidestepped the issue and settled on a simple but suboptimal solution. First, one defines the objective function (1) directly on the b-dimensional codes of each image (rather than on the hash function parameters) and optimizes it assuming continuous codes (in $\mathbb{R}^b$). Then, one binarizes the codes for each image. Finally, one learns a hash function given the codes. Optimizing the affinity-based loss function (1) can be done using spectral methods or nonlinear optimization as described above. Binarizing the codes has been done in different ways, from simply rounding them to $\{-1, +1\}$ using zero as threshold (Weiss et al., 2009; Zhang et al., 2010; Liu et al., 2011, 2012), to optimally finding a threshold (Liu et al., 2011; Strecha et al., 2012), to rotating the continuous codes so that thresholding introduces less error (Yu and Shi, 2003; Gong et al., 2013). Finally, learning the hash function for each of the b output bits can be considered as a binary classification problem, where the resulting classifiers collectively give the desired hash function, and can be solved using various machine learning techniques. Several works (e.g. Zhang et al., 2010; Lin et al., 2013, 2014) have used this approach, which does produce reasonable hash functions (in terms of retrieval measures such as precision and recall).

In order to do better, one needs to take into account during the optimization (rather than after the optimization) the fact that the codes are constrained to be binary. This implies attempting directly the discrete optimization of the affinity-based loss function over binary codes. This is a daunting task, since this is usually an NP-complete problem with Nb binary variables altogether, and practical applications could make this number as large as millions or beyond. Recent works have applied alternating optimization (with various refinements) to this, where one optimizes over a usually small subset of binary variables given fixed values for the remaining ones (Lin et al., 2013, 2014), and this did result in very competitive precision/recall compared with the state-of-the-art. This is still slow and future work will likely improve it, but as of now it provides an option to learn better binary codes.

Of the three-step suboptimal approach mentioned (learn continuous codes, binarize them, learn hash function), these works manage to join the first two steps and hence learn binary codes. Then, one learns the hash function given these binary codes. Can we do better? Indeed, in this paper we show that all elements of the problem (binary codes and hash function) can be incorporated in a single algorithm that optimizes jointly over them. Hence, by initializing it from binary codes from the previous approach, this algorithm is guaranteed to achieve a lower error and learn better hash functions. In fact, our framework can be seen as an iterated, corrected version of the two-step approach: learn binary codes *given the current hash function*, learn hash functions given codes, iterate (note the emphasis). The key to achieve this in a principled way is to use a recently proposed *method of auxiliary coordinates (MAC)* for optimizing "nested" systems, i.e., consisting of the composition of two or more functions or processing stages. MAC introduces new variables and constraints that cause decoupling between the stages, resulting in the mentioned alternation between learning the hash function and learning the binary codes. Section 3 reviews affinity-based loss functions, section 4 describes our MAC-based proposed framework and section 5 evaluates it in several supervised and unsupervised datasets, using linear and nonlinear hash functions.

2 Related work

Although one can construct hash functions without training data (Andoni and Indyk, 2008; Kulis and Grauman, 2012), we focus on methods that learn the hash function given a training set, since they perform better, and our emphasis is in optimization. The learning can be unsupervised, which attempts to preserve distances in the original space, or supervised, which in addition attempts to preserve label similarity. Many objective functions have been proposed to achieve this and we focus on affinity-based ones. These create an affinity matrix for a subset of training points based on their distances (unsupervised) or labels (supervised) and combine it with a loss function (Liu et al., 2012; Kulis and Darrell, 2009; Norouzi and Fleet, 2011; Lin et al., 2013, 2014). Some methods optimize this directly over the hash function. For example, Binary Reconstructive Embeddings (Kulis and Darrell, 2009) use alternating optimization over the weights of the hash functions. Supervised Hashing with Kernels (Liu et al., 2012) learns hash functions sequentially by considering the difference between the inner product of the codes and the corresponding element of the affinity matrix. Although many approaches exist, a common theme is to apply a greedy approach where one first finds codes using an affinity-based loss function, and then fits the hash functions to them (usually by training a classifier). The codes can be found by relaxing the problem and binarizing its solution (Weiss et al., 2009; Zhang et al., 2010; Liu et al., 2011), or by approximately solving for the binary codes using some form of alternating optimization, as in two-step hashing (Lin et al., 2013, 2014), or by using relaxation in other ways (Liu et al., 2012; Norouzi and Fleet, 2011).

3 Nonlinear embedding and affinity-based loss functions for binary hashing

The dimensionality reduction literature has developed a number of objective functions of the form (1) (often called "embeddings") where the low-dimensional projection $\mathbf{z}_n \in \mathbb{R}^b$ of each high-dimensional data point $\mathbf{x}_n \in \mathbb{R}^D$ is a free, real-valued parameter. The neighborhood information is encoded in the y_{nm} values (using labels in supervised problems, or distance-based affinities in unsupervised problems). A representative example is the elastic embedding (Carreira-Perpiñán, 2010), where $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form:

$$y^+_{nm} \left\| \mathbf{z}_n - \mathbf{z}_m \right\|^2 + \lambda y^-_{nm} \exp\left(- \left\| \mathbf{z}_n - \mathbf{z}_m \right\|^2\right),\ \lambda > 0 \tag{2}$$

where the first term tries to project true neighbors (having $y^+_{nm} > 0$) close together, while the second repels all non-neighbors' projections (having $y^-_{nm} > 0$) from each other. Laplacian Eigenmaps (Belkin and Niyogi, 2003) and Locally Linear Embedding (Roweis and Saul, 2000) result from replacing the second term above with a constraint that fixes the scale of $\mathbf{Z}$, which results in an eigenproblem rather than a nonlinear optimization, but also produces more distorted embeddings. Other objectives exist, such as t-SNE (van der Maaten and Hinton, 2008), that do not separate into functions of pairs of points. Optimizing nonlinear embeddings is quite challenging, but much progress has been done recently (Carreira-Perpiñán, 2010; Vladymyrov and Carreira-Perpiñán, 2012; van der Maaten, 2013; Yang et al., 2013; Vladymyrov and Carreira-Perpiñán, 2014). Although these models were developed to produce continuous projections, they have been successfully used for binary hashing too by truncating their codes (Weiss et al., 2009; Zhang et al., 2010) or using the two-step approach of (Lin et al., 2013, 2014).

Other loss functions have been developed specifically for hashing, where now $\mathbf{z}_n$ is a b-bit vector (where binary values are in $\{-1,+1\}$). For example (see a longer list in Lin et al., 2013), for Supervised Hashing with Kernels (KSH) $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form

$$(\mathbf{z}_n^T \mathbf{z}_m - b y_{nm})^2 \tag{3}$$

where y_{nm} is 1 if $\mathbf{x}_n$, $\mathbf{x}_m$ are similar and -1 if they are dissimilar. Binary Reconstructive Embeddings (Kulis and Darrell, 2009) uses

$$(\tfrac{1}{b} \left\| \mathbf{z}_n - \mathbf{z}_m \right\|^2 - y_{nm}^2)^2 \tag{4}$$

where $y_{nm} = \frac{1}{2} \left\| \mathbf{x}_n - \mathbf{x}_m \right\|^2$. Our approach can be applied to any of these loss functions, though we will focus on the KSH loss for simplicity.

4 Learning codes and hash functions using auxiliary coordinates

The optimization of the loss $\mathcal{L}(\mathbf{h})$ in eq. (1) is difficult because of the thresholded hash function, *which appears as the argument of the loss function L*. We use the recently proposed *method of auxiliary coordinates (MAC)* (Carreira-Perpiñán and Wang, 2012, 2014), which is a meta-algorithm to construct optimization algorithms for nested functions. This proceeds in 3 stages. First, we introduce new variables (the "auxiliary coordinates") as equality constraints into the problem, with the goal of unnesting the function. We can achieve this by introducing one binary vector $\mathbf{z}_n \in \{-1, 1\}$ for each point. This transforms the original, unconstrained problem into the following, constrained problem:

$$\min_{\mathbf{h},\mathbf{Z}} \sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \qquad \text{s.t.} \qquad \mathbf{z}_1 = \mathbf{h}(\mathbf{x}_1), \cdots, \mathbf{z}_N = \mathbf{h}(\mathbf{x}_N) \tag{5}$$

which is seen to be equivalent to (1) by eliminating $\mathbf{Z}$. We recognize as the objective function the "embedding" form of the loss function, except that the "free" parameters $\mathbf{z}_n$ are in fact constrained to be the deterministic outputs of the hash function $\mathbf{h}$.

Second, we solve the constrained problem using a penalty method, such as the quadratic-penalty or augmented Lagrangian (Nocedal and Wright, 2006). We discuss here the former for simplicity. We solve the following minimization problem (unconstrained again, but dependent on μ) while progressively increasing μ, so the constraints are eventually satisfied:

$$\min \mathcal{L}_Q(\mathbf{h}, \mathbf{Z}; \mu) = \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})\ + \mu \sum_{n=1}^{N} \left\| \mathbf{z}_n - \mathbf{h}(\mathbf{x}_n) \right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1, 1\}^b \tag{6}$$

Note that the quadratic penalty $\left\| \mathbf{z}_n - \mathbf{h}(\mathbf{x}_n) \right\|^2$ is proportional to the Hamming distance between the binary vectors $\mathbf{z}_n$ and $\mathbf{h}(\mathbf{x}_n)$.

Third, we apply alternating optimization over the binary codes $\mathbf{Z}$ and the hash function parameters $\mathbf{h}$. This results in iterating the following two steps (described in detail later):

- Optimize the binary codes $\mathbf{z}_1, \dots, \mathbf{z}_N$ given $\mathbf{h}$ (hence, given the output binary codes $\mathbf{h}(\mathbf{x}_1), \dots, \mathbf{h}(\mathbf{x}_N)$ for each of the N images). This can be seen as a *regularized binary embedding*, because the projections $\mathbf{Z}$ are encouraged to be close to the hash function outputs $\mathbf{h}(\mathbf{X})$. Here, we try two different approaches (Lin et al., 2013, 2014) with some modifications.

- Optimize the hash function $\mathbf{h}$ given binary codes $\mathbf{Z}$. This reduces to training b binary classifiers using $\mathbf{X}$ as inputs and $\mathbf{Z}$ as targets.

Notice this is very similar to the two-step (TSH) approach of Lin et al. (2013), except that the latter learns the codes $\mathbf{Z}$ in isolation, rather than given the current hash function, so iterating the two-step approach would change nothing, and it does not optimize the loss $\mathcal{L}$. More precisely, TSH corresponds to optimizing $\mathcal{L}_Q$ for $\mu \to 0$. In practice, we start from a very small value of μ (hence, initialize MAC from the result of TSH), and increase μ slowly while optimizing $\mathcal{L}_Q$, until the equality constraints are satisfied, i.e., $\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$

```
input $\mathbf{X}_{D\times N} = (\mathbf{x}_1,\dots,\mathbf{x}_N)$, $\mathbf{Y}_{N\times N} = (y_{nm})$, $b \in \mathbb{N}$
Initialize $\mathbf{Z}_{b\times N} = (\mathbf{z}_1,\dots,\mathbf{z}_N) \in \{0,1\}^{bN}$
for $\mu = 0 < \mu_1 < \dots < \mu_\infty$
  for $i = 1,\dots,b$                                                h step
    $h_i \leftarrow$ fit hash function to $(\mathbf{X}, \mathbf{Z}_{\cdot i})$
  repeat                                                             Z step
    for $i = 1,\dots,b$
      $\mathbf{Z}_{\cdot i} \leftarrow$ approximate minimizer of $\mathcal{L}_Q(\mathbf{h},\mathbf{Z};\mu)$ over $\mathbf{Z}_{\cdot i}$
  until no change in $\mathbf{Z}$ or maxit cycles ran
  if $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ then stop
return $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$
```

Figure 1: MAC algorithm to optimize an affinity-based loss function for binary hashing.

for $n = 1,\dots,N$. One can prove this occurs for a finite value of μ if the $\mathbf{Z}$ and $\mathbf{h}$ steps are exact. Hence, unlike in quadratic-penalty methods for continuous problems, we do not need to drive $\mu \to \infty$, so we avoid the progressive ill-conditioning that affects penalty methods. During the optimization, for intermediate μ, the parameters $\mathbf{Z}$ and $\mathbf{h}$ can wander through regions of the space that violate the constraints, i.e., codes that do not match the output of the hash function. This can potentially lead to better local optima. Fig. 1 gives the overall MAC algorithm to learn a hash function by optimizing an affinity-based loss function.

4.1 h step

Given the binary codes $\mathbf{z}_1,\dots,\mathbf{z}_N$, since $\mathbf{h}$ does not appear in the first term of $\mathcal{L}_Q$, this simply involves finding a hash function $\mathbf{h}$ that minimizes

$$\min_{\mathbf{h}} \sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 = \sum_{i=1}^{b} \min_{h_i} \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2 \tag{7}$$

where $z_{ni} \in \{-1,1\}$ is the ith bit of the binary vector $\mathbf{z}_n$. Hence, we can find b one-bit hash functions in parallel and concatenate them into the b-bit hash function. Each of these is a binary classification problem using the number of misclassified patterns as loss. This allows us to use a regular classifier for $\mathbf{h}$, and even to use a simpler surrogate loss (such as the hinge loss), since this will also enforce the constraints eventually (as μ increases). For example, we can fit an SVM by optimizing the margin plus the slack and using a high penalty for misclassified patterns. We discuss other classifiers in the experiments.

4.2 Z step

Although the MAC technique has significantly simplified the original problem, the step over $\mathbf{Z}$ is still complex. This involves finding the binary codes given the hash function $\mathbf{h}$, and it is an NP-complete problem in Nb binary variables. Fortunately, some recent works have proposed practical approaches for this problem based on alternating optimization: a quadratic surrogate method (Lin et al., 2013), and a GraphCut method (Lin et al., 2014). In both cases, this would correspond to the first step in the two-step hashing of Lin et al. (2013).

In both the quadratic surrogate and the GraphCut method, the starting point is to apply alternating optimization over the ith bit of all points given the remaining bits are fixed for all points (for $i = 1,\dots,b$), and to solve the optimization over the ith bit approximately. We describe this next for each method. We start by describing each method in their original form (which applies to the loss function over binary codes, i.e., the first term in $\mathcal{L}_Q$), and then we give our modification to make it work with our $\mathbf{Z}$ step objective (the regularized loss function over binary codes, i.e., the complete $\mathcal{L}_Q$).

4.2.1 Solution using a quadratic surrogate method (Lin et al., 2013)

This is based on the fact that any loss function that depends on the Hamming distance of two binary variables can be equivalently written as a quadratic function of those two binary variables (Lin et al., 2013). Since this is the case for every term $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ (because only the ith bit in each of $\mathbf{z}_n$ and $\mathbf{z}_m$ is free), we can write the first term in $\mathcal{L}_Q$ as a binary quadratic problem. We now consider the second term (on μ) as well. (We use a similar notation as that of Lin et al., 2013.) The optimization for the ith bit can be written as:

$$\min_{\mathbf{z}_{(i)}} \sum_{n,m=1}^{N} l_i(z_{ni}, z_{mi}) + \mu \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2 \tag{8}$$

where $l_i = L(z_{ni}, z_{mi}, \bar{\mathbf{z}}_n, \bar{\mathbf{z}}_m; y_{nm})$ is the loss function defined on the ith bit, z_{ni} is the ith bit of the nth point, $\bar{\mathbf{z}}_n$ is a vector containing the binary codes of the nth point except the ith bit, and $h_i(\mathbf{x}_n)$ is the ith bit of the binary code of the nth point generated by the hash function $\mathbf{h}$. Lin et al. (2013) show that $l(z_1, z_2)$ can be replaced by a binary quadratic function

$$l(z_1, z_2) = \frac{1}{2} z_1 z_2 \big(l^{(11)} - l^{(-11)}\big) + \text{constant} \tag{9}$$

as long as $l(1,1) = l(-1,-1) = l^{(11)}$ and $l(1,-1) = l(-1,1) = l^{(-11)}$, where $z_1, z_2 \in \{-1, 1\}$. Equation (9) helps us to rewrite the optimization (8) as the following:

$$\min_{\mathbf{z}_{(i)}} \sum_{n,m=1}^{N} \frac{1}{2} z_{ni} z_{mi} \big(l^{(11)} - l^{(-11)}\big) + \mu \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2. \tag{10}$$

By defining $a_{nm} = \big(l^{(11)}_{(inm)} - l^{(-11)}_{(inm)}\big)$ as the (n,m) element of a matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and ignoring the coefficients, we have the following optimization problem:

$$\min_{\mathbf{z}_{(i)}} \mathbf{z}_{(i)}^T \mathbf{A} \mathbf{z}_{(i)} + \mu \left\|\mathbf{z}_{(i)} - \mathbf{h}_i(\mathbf{X})\right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_{(i)} \in \{-1, +1\}^N$$

where $\mathbf{h}_i(\mathbf{X}) = (h_i(\mathbf{x}_1), \dots, h_i(\mathbf{x}_N))^T$ is a vector of length N (one bit per data point). Both terms in the above minimization are quadratic on binary variables. This is still an NP-complete problem (except in special cases), and we approximately solve it by relaxing it to a continuous quadratic program (QP) over $\mathbf{z}_{(i)} \in [-1, 1]^N$ and binarizing its solution. In general, the matrix $\mathbf{A}$ is not positive definite and the relaxed QP is not convex, so we need an initialization. (However, the term on μ adds $\mu\mathbf{I}$ to $\mathbf{A}$, so even if $\mathbf{A}$ is not positive definite, $\mathbf{A} + \mu\mathbf{I}$ will be positive definite for large enough μ, and the QP will be convex.) We construct an initialization by converting the binary QP into a binary eigenproblem:

$$\min_{\boldsymbol{\alpha}} \boldsymbol{\alpha}^T \mathbf{B} \boldsymbol{\alpha} \qquad \text{s.t.} \qquad \alpha_0 = 1, \quad \mathbf{z}_{(i)} \in \{-1, 1\}^N, \quad \boldsymbol{\alpha} = \begin{pmatrix} \mathbf{z}_{(i)} \\ \alpha_0 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \mathbf{A} & -\frac{\mu}{2}\mathbf{h}_i(\mathbf{X}) \\ -\frac{\mu}{2}\mathbf{h}_i(\mathbf{X})^T & 0 \end{pmatrix}. \tag{11}$$

To solve this problem we use spectral relaxation, where the constraints $\mathbf{z}_{(i)} \in \{-1, 1\}^N$ and $z_{i+1} = 1$ are relaxed to $\|\boldsymbol{\alpha}\| = N + 1$. The solution to this problem is the eigenvector corresponding to the smallest eigenvalue of $\mathbf{B}$. We use the truncated eigenvector as the initialization for minimizing the relaxed, bound-constrained QP:

$$\min_{\mathbf{z}_{(i)}} \mathbf{z}_{(i)}^T \mathbf{A} \mathbf{z}_{(i)} + \mu \left\|\mathbf{z}_{(i)} - \mathbf{h}_i(\mathbf{X})\right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_{(i)} \in [-1, 1]^N.$$

which we solve using L-BFGS-B (Zhu et al., 1997).

As noted above, the $\mathbf{Z}$ step is an NP-complete problem in general, so we cannot expect to find the global optimum. It is even possible that the approximate solution could increase the objective over the previous iteration's $\mathbf{Z}$ (this is likely to happen as the overall MAC algorithm converges). If that occurs, we simply skip the update, in order to guarantee that we decrease monotonically on $\mathcal{L}_Q$, and avoid oscillating around a minimum.

4.2.2 Solution using a GraphCut algorithm (Lin et al., 2014)

To optimize over the ith bit (given all the other bits are fixed), we have to minimize eq. (10). In general, this is an NP-complete problem over N bits (the ith bit for each image), with the form of a quadratic function on binary variables. We can apply the GraphCut algorithm (Boykov and Kolmogorov, 2003, 2004; Kolmogorov and Zabih, 2003), as proposed by the FastHash algorithm of Lin et al. (2014). This proceeds as follows. First, we assign all the data points to different, possibly overlapping groups (blocks). Then, we minimize the objective function over the binary codes of the same block, while all the other binary codes are fixed, then proceed with the next block, etc. (that is, we do alternating optimization of the bits over the blocks). Specifically, to optimize over the bits in block $\mathcal{B}$, we define $a_{nm} = \big(l^{(11)}_{(inm)} - l^{(-11)}_{(inm)}\big)$ and, ignoring the constants, we can rewrite equation (10) as:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \sum_{n\in\mathcal{B}} \sum_{m\in\mathcal{B}} a_{nm} z_{ni} z_{mi} + 2 \sum_{n\in\mathcal{B}} \sum_{m\notin\mathcal{B}} a_{nm} z_{ni} z_{mi} - \mu \sum_{n\in\mathcal{B}} z_{ni} h_i(\mathbf{x}_n).$$

We then rewrite this equation in the standard form for the GraphCut algorithm:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \sum_{n\in\mathcal{B}} \sum_{m\in\mathcal{B}} v_{nm} z_{ni} z_{mi} + \sum_{n\in\mathcal{B}} u_{nm} z_{ni} z_{mi} \qquad \text{where} \qquad v_{nm} = a_{nm}, \quad u_{nm} = 2 \sum_{m\notin\mathcal{B}} a_{nm} z_{mi} - \mu h_i(\mathbf{x}_n). \tag{12}$$

To minimize the objective function using the GraphCut algorithm, the blocks have to define a submodular function. For the objective functions that we explained in the paper, this can be easily achieved by putting points with the same label in one block (Lin et al., 2014 give a simple proof of this).

Unlike in the quadratic surrogate method, using the GraphCut algorithm with alternating optimization on blocks defining submodular functions is guaranteed to find a $\mathbf{Z}$ that has a lower or equal objective value that the initial one, and therefore to decrease monotonically $\mathcal{L}_Q$.

5 Experiments

We have tested our framework with several combinations of loss function, hash function, both supervised and unsupervised datasets, and comparing with several state-of-the-art hashing methods. Here we report a representative subset to show the flexibility of the approach. We focus mostly on the KSH loss function (3) (Liu et al., 2012), but also show results for the Elastic Embedding loss function (Carreira-Perpiñán, 2010). We use both the quadratic surrogate and the GraphCut method for the $\mathbf{Z}$ step in MAC. We use linear and kernel SVM hash functions (for each bit). To train the linear SVM, we use LIBLINEAR (Fan et al., 2008). To train the kernel SVM, we use 500 radial basis functions with centers given by a random subset of the training points, and apply a linear SVM to their output. (Computationally, this is fast because we can use a constant Gram matrix. Using as hash function a kernel SVM trained with LIBSVM gave similar results, but it is much slower because the support vectors change when the labels change.)

We use the following datasets (all using the Euclidean distance in feature space): (1) SIFT10K (Jégou et al., 2011) contains $N = 10\,000$ images for training and 100 images for test, each represented by $D = 128$ SIFT features. (2) NUS-WIDE-LITE (Chua et al., 2009) contains $N = 27\,807$ high resolution images for training and 27 808 for test, each represented by $D = 128$ wavelet features. (3) CIFAR (Krizhevsky, 2009) contains 60 000 images in 10 classes. We extract $D = 320$ GIST features (Oliva and Torralba, 2001) from each image. We use 58 000 images for training and 2 000 images for test. (4) COIL-20 (Nene et al., 1996) contains 1 440 32×32 grayscale images, using the $D = 1\,024$ pixel grayscales as features. Since the dataset is small, we use the whole dataset for both training and test. (5) USPS (Hull, 1994) contains 9 298 16×16 grayscale images of handwritten digits from 0 to 9. We use $N = 7\,291$ images for training and 2 007 for test, using the $D = 256$ pixel grayscales as features. (6) Pen-Digits contains 10 992 images of digits from 0 to 9. We use $N = 8\,992$ images for training and 2 000 for test, each represented by a $D = 256$ feature vector.

We report precision/recall on the test set using the following ground truth (set of true neighbors in original space): the K nearest neighbors in unsupervised datasets, and all the training points with the same label in supervised datasets. The set of retrieved neighbors for a test point is either the k nearest points in the Hamming space, or the points within a Hamming distance r (we report zero precision if no point lies inside Hamming distance r).

Unless otherwise indicated, all our experiments use the KSH loss, and we use the following schedule for the penalty parameter μ in the MAC algorithm (regardless of the hash function type or dataset): we set μ to 0 in the first iteration (i.e., we initialize MAC from the result of two-step hashing; Lin et al., 2013) and increase it by 1 after each iteration (i.e., after a $\mathbf{Z}$ and $\mathbf{h}$ step), until the codes stop changing and the algorithm stops.

Our experiments show that the MAC algorithm indeed finds hash functions with a significantly and consistently lower objective function value than rounding or two-step approaches; and that it outperforms other unsupervised and supervised state-of-the-art algorithms on different datasets. We also find, in agreement with Lin et al. (2014), that using the GraphCut method within the $\mathbf{Z}$ step is preferable to using the quadratic surrogate method: the overall MAC algorithm is faster and results in better optima.

5.1 The MAC algorithm finds better optima

The goal of this paper is not to introduce a new loss or hash function, but to describe a generic framework to construct algorithms that optimize a given combination thereof. We illustrate its effectiveness here using two datasets, with different sizes of retrieved neighbor sets, and using 8 to 32 bits. In all cases, the MAC algorithm achieves a better hash function both in terms of the loss and of the precision. We compare 3 ways of optimizing the KSH loss function (3): KSH (Liu et al., 2012), two-step hashing (TSH) (Lin et al., 2013) and MAC using the quadratic surrogate method in the $\mathbf{Z}$ step.

For the SIFT10K dataset we train all methods with a subset of $3,000$ points. All methods use linear SVMs as hash function, and MAC and TSH also use kernel SVMs. For the NUS-WIDE-LITE dataset we show MAC and TSH with a linear hash function. Fig. 2(top) shows the loss function for all the methods over iterations of the MAC algorithm (KSH and TSH do not iterate). It is clear that the MAC algorithm consistently reduces the loss function more than KSH and TSH in all cases. The plots also show MAC's nonmonotonic decrease of the loss function. This is because MAC operates in the augmented space of the codes and the hash function, trading off decreasing the loss with enforcing the constraints as μ increases. While it always takes steps that decrease the quadratic-penalty objective $\mathcal{L}_Q$, each step need not decrease the loss $\mathcal{L}$ (though it typically does). Fig. 2(bottom) shows that reducing the loss nearly always translates into better precision by a significant margin.

5.2 Comparison with hashing methods in unsupervised datasets

We compare MAC (using the quadratic surrogate method in the $\mathbf{Z}$ step) with thresholded PCA (tPCA), Two-Step Hashing (TSH) (Lin et al., 2013), Iterative Quantization (ITQ) (Gong et al., 2013), Spectral Hashing (SH) (Weiss et al., 2009), Kernelized Locality-Sensitive Hashing (KLSH) (Kulis and Grauman, 2012), AnchorGraph Hashing (AGH) (Liu et al., 2011), and Spherical Hashing (SPH) (Heo et al., 2012). MAC and TSH use the KSH loss function (3). We create its affinity matrix y_{nm} by declaring as similar points for each training point $\mathbf{x}_n$ its 100 true nearest neighbors, and as dissimilar points a random subset of 100 among the remaining points. MAC takes 5' to train a subset of 3 000 points for $b = 32$ bits. (We do observe that making the affinity matrix less sparse by adding more dissimilar entries achieves better precision, but the runtime increases; using 1 000 dissimilar entries, the MAC algorithm takes $\approx$ 15' to train.)

Fig. 3 shows the results in several datasets. In NUS-WIDE-LITE we use all 27 807 training points to train each method (including MAC and TSH). We use a linear SVM as hash function for MAC and TSH. We considered as ground truth the $K = 500$ true nearest neighbors of the query image, and retrieve either $k = 50$ nearest neighbors or the neighbors within Hamming distance r ($r = 1$ to 3). The plots show that MAC is always better than TSH by a good margin. When b is large and r is small, the precision of the different methods decreases. In practice, one would simply increase r to retrieve sufficient neighbors.

COIL-20 and SIFT10K are smaller datasets. For COIL-20 we use all 1 440 points to train each method. For SIFT10K, we train MAC and TSH on a subset of 3 000 training set points and the other methods on all the training points. We considered as ground truth the $K = 50$ and $K = 100$ true nearest neighbors for COIL-20 and SIFT10K, respectively. The retrieved set contains the k nearest neighbors of each point, wehere k is equal to the number of points K in the ground truth set. The plots show that MAC outperforms the comparison methods on this small datasets as well. In SIFT10K, ITQ achieves better precision than TSH for $b = 24$ and 32 bits while MAC is better than or as good as ITQ in these cases.

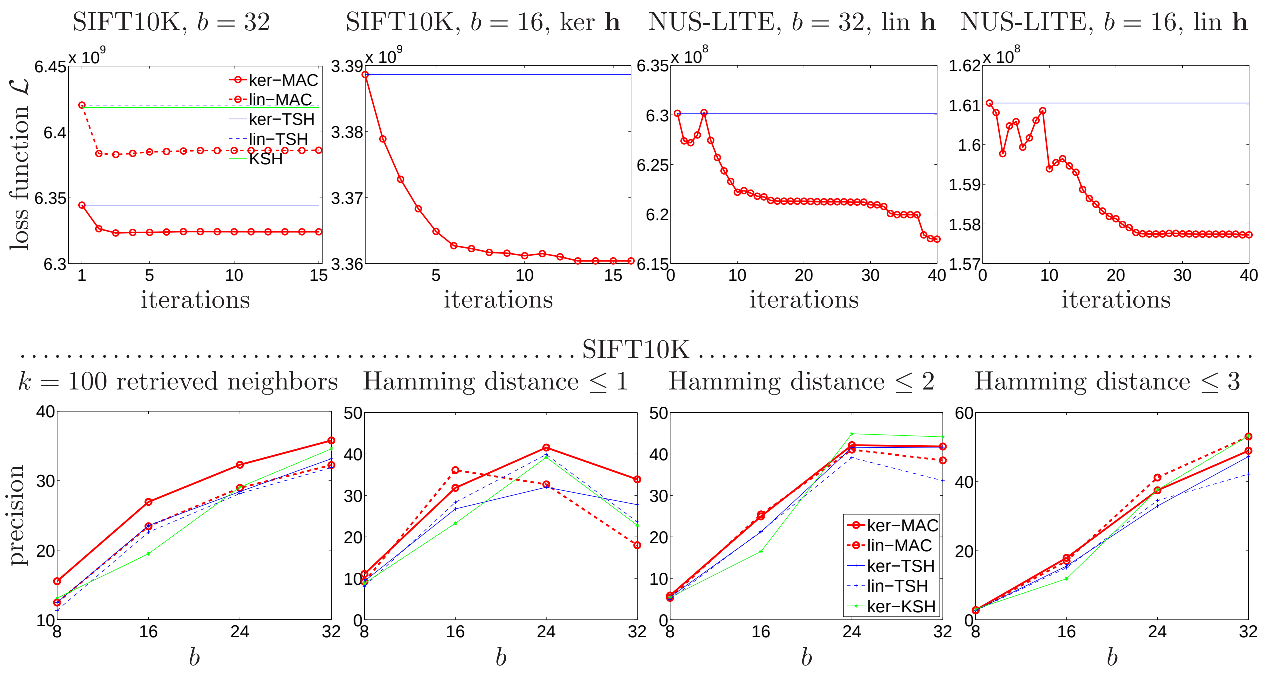

Figure 2: The MAC algorithm finds better optima. Here, MAC uses a quadratic surrogate in the $\mathbf{Z}$ step. *Top*: value of the KSH loss function $\mathcal{L}$ for Two-Step Hashing (TSH), Supervised Hashing with Kernels (KSH) and MAC (over iterations). *Bottom*: precision in SIFT10K using $b = 8$ to 32 bits. Ground truth: $K = 100$ nearest images to the query image in the training set. Retrieved neighbors: $k = 100$ nearest images to or images at Hamming distance $\leq r = 1$, 2 and 3 of the query, searching the training set binary codes.

5.3 Comparison with hashing methods in supervised datasets

We compare MAC (using the quadratic surrogate method in the $\mathbf{Z}$ step) with Two-Step Hashing (TSH) (Lin et al., 2013), Supervised Hashing with Kernels (KSH) (Liu et al., 2012), Iterative Quantization (ITQ) with supervised embedding (Gong et al., 2013), Supervised Binary Reconstructive Embeddings (BRE) (Kulis and Darrell, 2009) and Supervised Self-Taught Hashing (STH) (Zhang et al., 2010). MAC and TSH use the KSH loss function (3). We create the affinity matrix y_{nm} for MAC and TSH based on the labels of the points. For each training point $\mathbf{x}_n$, the set of similar and dissimilar points are a random subset of all the points with the same or different label as $\mathbf{x}_n$, respectively.

Fig. 4 show results in several datasets. In USPS and Pen-Digits. both of which are small datasets, all methods achieve good results (over 80% precision with only 12 bits), but MAC is again the overall winner, with a particularly large margin for smaller bit values. In CIFAR, we train all the methods using a subset of 3 000 points. We consider as set of retrieved neighbors either $k = 100$ nearest neighbors or neighbors within Hamming distance $r = 1$, 2 and 3 (top), or we fix the number of bits to $b = 32$ and change the number of points in the retrieved set from 1 to 1 000 (bottom). In almost all cases, MAC achieves better precision than the other methods, often by a considerable margin.

5.4 Comparison using code utilization

Fig. 5 shows the results (for all methods on CIFAR and SIFT10K) in effective number of bits b_{eff}. This is a measure of code utilization of a hash function introduced by Carreira-Perpiñán and Raziperchikolaei (2015), defined as the entropy of the code distribution. That is, given the N codes $\mathbf{z}_1, \dots, \mathbf{z}_N \in \{0,1\}^b$ for the training set, we consider them as samples of a distribution over the 2^b possible codes. The entropy of this distribution, measured in bits, is between 0 (when all N codes are equal) and $\min(b, \log_2 N)$ (when all N codes are distributed as uniformly as possible). We do the same for the test set. Although code utilization correlates to some extent with precision/recall when ranking different methods, a large b_{eff} does not guarantee a good hash function, and indeed, tPCA (which usually achieves a low precision compared to the state-of-the-

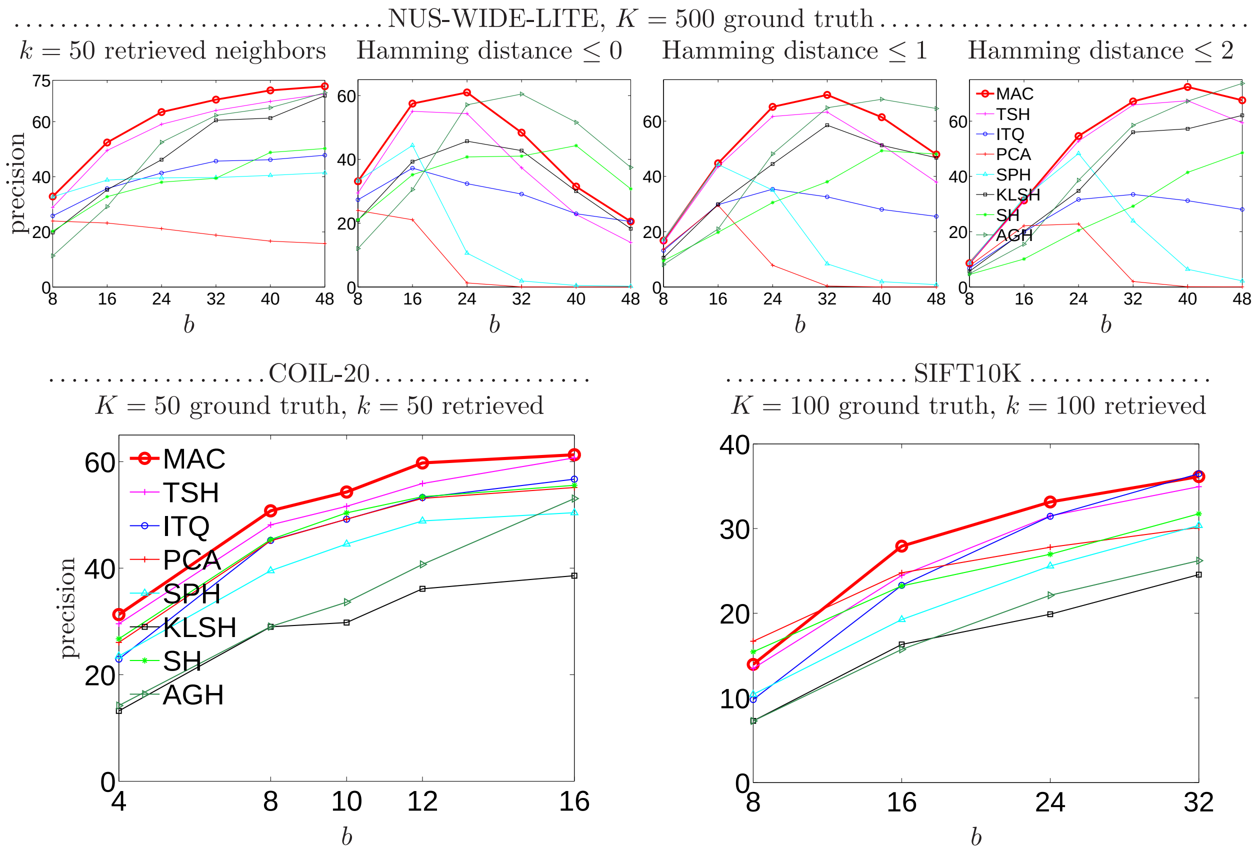

Figure 3: Results in unsupervised datasets. MAC uses a quadratic surrogate in the $\mathbf{Z}$ step. Ground truth: K nearest images to the query image in the training set. Retrieved neighbors: k nearest images to or images at Hamming distance $\leq r$ of the query, searching the training set binary codes. *Top*: precision in NUS-WIDE-LITE using $b = 8$ to 32 bits. *Bottom*: precision in COIL-20 using $b = 4$ to 16 bits (*left*), and SIFT10K using $b = 8$ to 32 bits (*right*).

art) typically achieves the largest b_{eff}; see the discussion in Carreira-Perpiñán and Raziperchikolaei (2015). However, a large b_{eff} does indicate a better use of the available codes (and fewer collisions if $N < 2^b$), and b_{eff} has the advantage over precision/recall that it does not depend on any user parameters (such as ground truth size or retrieved set size), so we can compare all binary hashing methods with a single number b_{eff} (for a given number of bits b). It is particularly useful to compare methods that are optimizing the same objective function. With this in mind, and noting that TSH and MAC both optimize the same objective, fig. 5 shows that MAC has a consistently larger b_{eff}, indicating that not only does MAC achieve a better objective function value and precision/recall than TSH, but also a better code utilization.

Earlier we noted that, if using a small Hamming distance r in the retrieved set, the precision drops for some methods as b increases. We can see that these are the methods that have largest b_{eff} (for CIFAR, these are BRE, KSH and MAC). These methods are making good use of the available codes, having few collisions, so they need a larger Hamming distance r to find neighbors. As b increases, their b_{eff} stagnates because it is limited by $\log_2 N$ (where N is the number of points, which at some point is small compared with the number of possible codes 2^b). If the training or test set was larger, we would expect the precision and b_{eff} to continue to increase for those same values of b.

5.5 Using GraphCut in the Z step of the MAC algorithm

Now we use the GraphCut instead of the quadratic surrogate in the $\mathbf{Z}$ step of the MAC algorithm. We use the CIFAR dataset and train both a linear and a kernel SVM as the hash function. We compare our

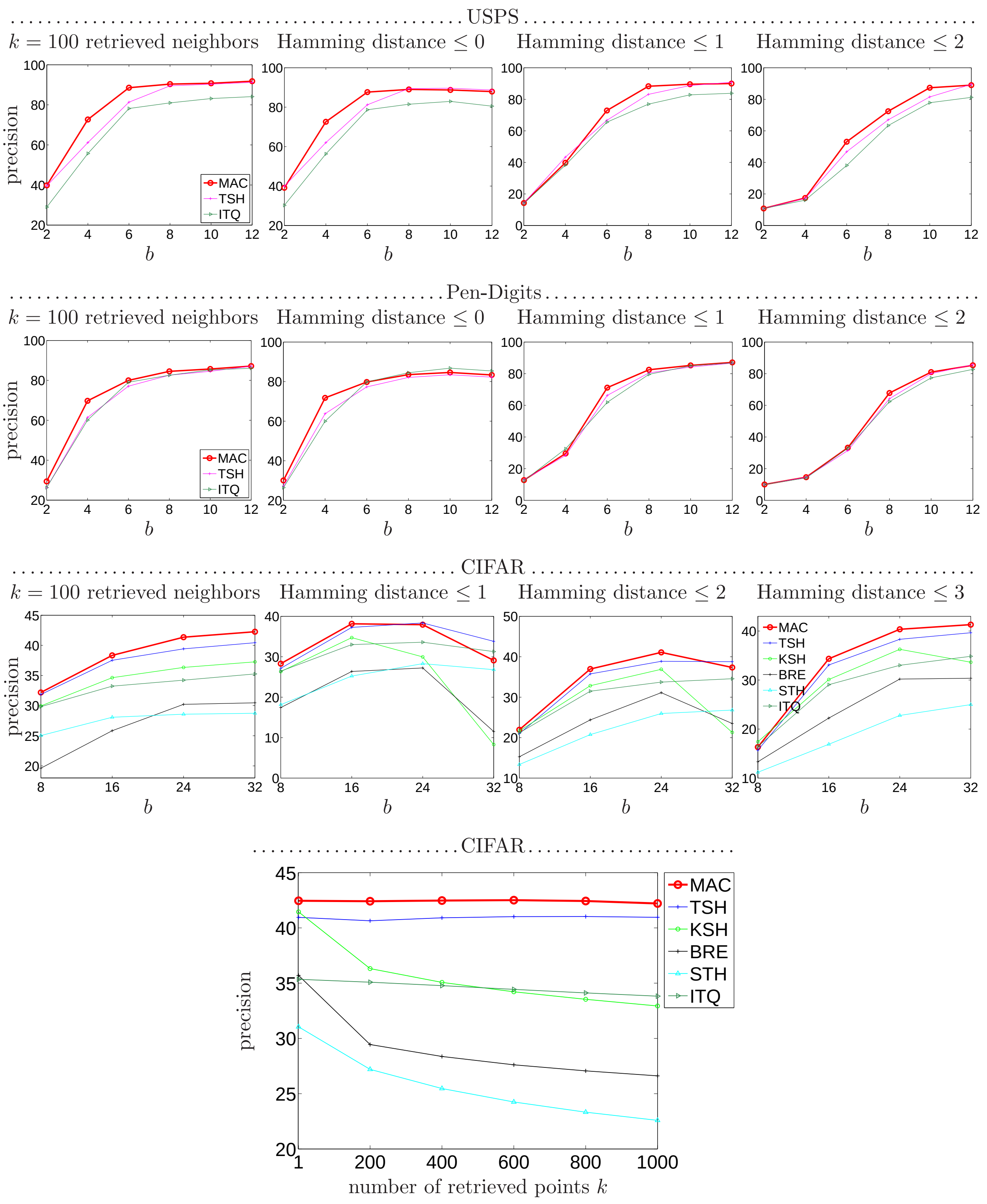

Figure 4: Results in supervised datasets. MAC uses a quadratic surrogate in the $\mathbf{Z}$ step. Ground truth: points with the same label as the query image in the training set. Retrieved neighbors: k nearest images to or images at Hamming distance $\leq r$ of the query, searching the training set binary codes. *Row 1*: precision in USPS using $b = 2$ to 12 bits. *Row 2*: precision in Pen-Digits using $b = 2$ to 12 bits. *Row 3*: precision in CIFAR using $b = 8$ to 32 bits. *Bottom panel*: precision in CIFAR using $b = 32$ bits, retrieving as neighbors the k nearest images to the query, searching the training set binary codes, for $k = 1$ to 1 000.

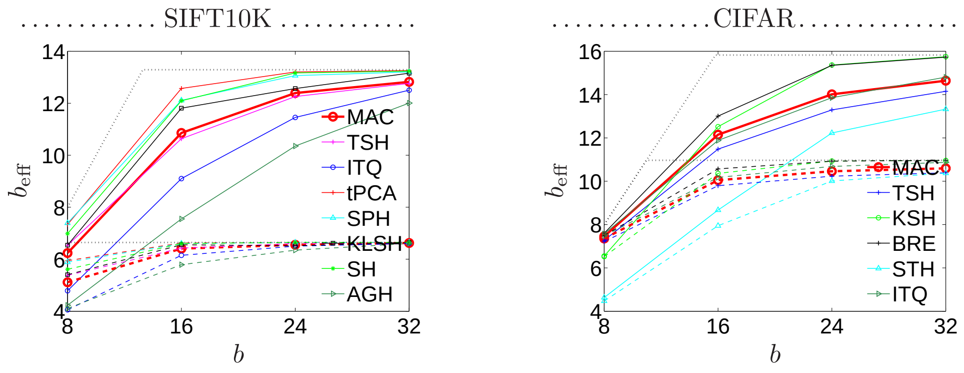

Figure 5: Code utilization in effective number of bits b_{eff} (entropy of code distribution) of different hashing algorithms, using $b = 8$ to 32 bits, for the SIFT10K (*left*) and CIFAR (*right*) datasets. The plots correspond to the codes obtained by the algorithms in figures 3 (bottom right) and 4 (row 3), respectively. The hashing algorithms are color-coded, with solid lines for the training set and dashed lines for the test set. The two diagonal-horizontal black dotted lines give the upper bound (maximal code utilization) $\min(b, \log_2 N)$ on b_{eff} of any algorithm for the training and test sets (where N is the size of the training or test set).

method with FastHash (Lin et al., 2014), which (like two-step hashing; Lin et al., 2013) proceeds in a greedy, suboptimal way: first, find binary codes using the GraphCut algorithm described earlier; then, fit a hash function to them, and return this as the final hash function.

In our first experiment, we train our algorithm using 10 000 randomly selected images of the CIFAR dataset with the KSH objective function (Liu et al., 2012). To create the affinity matrix, for each point we randomly select 100 points having its same label and 100 points having a dissimilar label. We set the penalty parameter μ to 0 in the first iteration and we increase it by 1 after each iteration. Fig. 6 shows the results. The first row shows the objective function (1) over the iterations of our MAC algorithm in the training set (for different numbers of bits). The second row shows the precision achieved in the test set for the resulting hash function. Note that the FastHash algorithm corresponds to $\mu \to 0$, i.e., the result in our first iteration. As we expected, our MAC algorithm outperforms FastHash in all cases in the objective function value, and in almost all cases in precision. The only case where we do worse is for $b = 32$ bits when retrieving neighbors at a Hamming distance $r \leq 2$ (which is a small distance for a large number of bits). This simply happens because many of the test points have no neighbors within Hamming distance 2 and are assigned zero precision. If increasing r to 3 or 4 in order to retrieve more neighbors, we consistently improve over FastHash. This indicates that our algorithm is better avoiding collisions in binary space by making better use of the available codes. In the last row, we systematically try different numbers of retrieved points and report the precision, in two situations: with the CIFAR subset using both linear and kernel SVMs; and with the entire CIFAR dataset ($N = 58\,000$, the runtime was around 4 hours). In all cases, our MAC algorithm consistently gives better precision than FastHash, and the precision is larger than when training on the smaller subset of $N = 10\,000$ images.

Lin et al. (2014) noted that the GraphCut method works better than the quadratic relaxation of Lin et al. (2013) and our results confirm this when trying both methods within our $\mathbf{Z}$ step. GraphCut works generally better because it gives us the opportunity to use larger subsets of points for training. The relaxed solution scales worse as the number of nonzero affinities grows and limits us to using small subsets of data points to learn the hash function. Small subsets may not represent the relation between the points in the original dataset well enough. In this case, decreasing the objective function does not necessarily lead to (significant) increases in the precision. Our experiments show that we can overcome this problem by selecting a large amount of negative entries in the affinity matrix. If using the GraphCut method, we do not have this problem. We can select a larger subset of the dataset (10 000 points, for example) to learn the hash function. Since the subset can be larger, the number of positive and negative entries of the affinity matrix can be smaller.

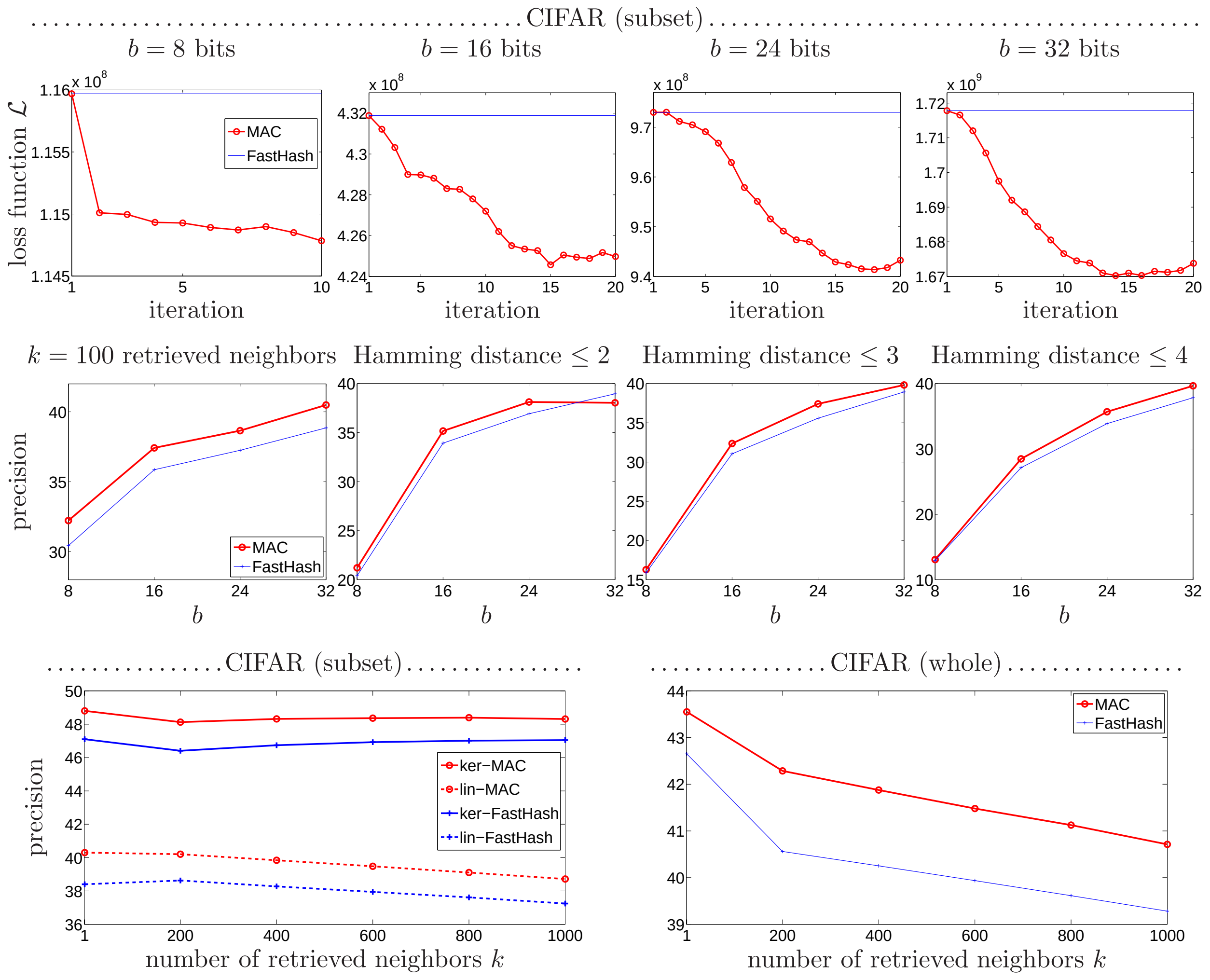

Figure 6: Using GraphCut in the $\mathbf{Z}$ step of the MAC algorithm. *Top*: value of the KSH loss function $\mathcal{L}$ (top) for FastHash and our MAC algorithm (over iterations), and precision (%) for the resulting hash function using $b = 8$ to 32 bits). We train a linear SVM hash function using a subset of the CIFAR dataset containing 10 000 images. Ground truth: points with the same label as the query image in the training set. Retrieved neighbors: $k = 100$ nearest images to or images at Hamming distance $\leq r = 2$, 3 and 4 of the query, searching the training set binary codes. *Bottom*: precision over a range of retrieved neighbors $k = 1$ to 1 000 with $b = 32$ bits, using the CIFAR subset with both linear and kernel SVM hash functions (*left*), and using the entire CIFAR training set ($N = 58\,000$ images) with a linear SVM.

5.6 Using the Elastic Embedding loss function

We now use as loss function L in eq. (1) the Elastic Embedding (EE) (Carreira-Perpiñán, 2010), originally proposed as a continuous nonlinear embedding method, whose objective function is in eq. (2). We set the λ parameter (which trades off attraction between true neighbors and repulsion between non-neighbors) to 500. We use the GraphCut method for the $\mathbf{Z}$ step of the MAC algorithm. In the first iteration, we set μ to zero, then increase it to 50, then increase by 10 after each iteration. Other settings are as in the previous experiment with GraphCut. Fig. 7 shows the result. As before, our MAC algorithm performs better than FastHash (both of which using the same hash function) and achieves better precision. This demonstrates that the MAC algorithm can always improve over the two-step hashing or FastHash, over a wide range of conditions: choice of loss or hash function, and choice of number of bits. (As before, the lower precision for $b = 32$ bits with Hamming distance ≤ 2 is due to many of the test points having no neighbors within that small Hamming distance, and one should use a larger one.)

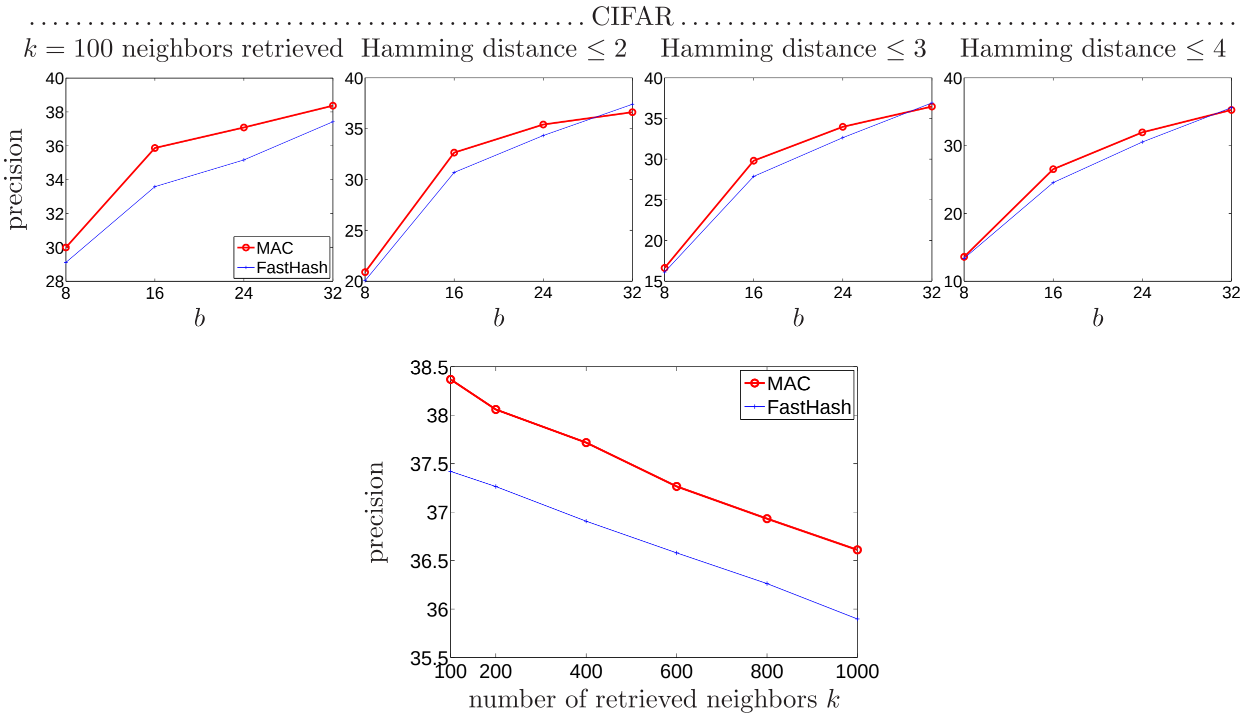

Figure 7: Results using the Elastic Embedding loss function. MAC uses GraphCut in the $\mathbf{Z}$ step. Same experiment setup as in fig. 6 with a linear SVM. *Top*: precision (%) using $b = 8$ to 32 bits. Ground truth: points with the same label as the query image in the training set. Retrieved neighbors: $k = 100$ nearest images to or images at Hamming distance $\leq r = 2$, 3 and 4 of the query, searching the training set binary codes. *Bottom*: precision using $b = 32$ bits over a range of retrieved neighbors.

6 Discussion

The two-step approach of Two-Step Hashing (Lin et al., 2013) and FastHash (Lin et al., 2014) is a significant advance in finding good codes for binary hashing, but it also causes a maladjustment between the codes and the hash function, since the codes were learned without knowledge of what hash function would use them. Ignoring the interaction between the loss and the hash function limits the quality of the results. For example, a linear hash function will have a harder time than a nonlinear one at learning such codes. In our algorithm, this tradeoff is enforced gradually (as μ increases) in the $\mathbf{Z}$ step as a regularization term (eq. (6)): it finds the best codes according to the loss function, but makes sure they are close to being realizable by the current hash function. Our experiments demonstrate that significant, consistent gains are achieved in both the loss function value and the precision/recall in image retrieval over the two-step approach.

A similar, well-known situation arises in feature selection for classification (Kohavi and John, 1998). The best combination of classifier and features will result from jointly minimizing the classification error with respect to both classifier and features (the "wrapper" approach), rather than first selecting features according to some criterion and then using them to learn a particular classifier (the "filter" approach).

The method of auxiliary coordinates algorithmically decouples (within each iteration) the two elements that make up a binary hashing model: the hash function and the loss function. Both elements act in combination to produce a function that maps input patterns to binary codes so that they represent neighborhood in input space, but they play distinct roles. The hash function role is to map input patterns to binary codes. The loss function role is to assign binary codes to input patterns in order to preserve neighborhood relations, regardless of how easy it is for a mapping to produce such binary codes. By itself, the loss function would produce a nonparametric hash function for the training set with the form of a table of (image,code) pairs. However, the hash function and the loss function cannot act independently, because the objective function depends on both. The optimal combination of hash and loss is difficult to obtain, because of the nonlinear and discrete nature of the objective. Several previous optimization attempts for binary hashing first find codes that optimize the loss, and then fit a hash function to them, thus imposing a strict, suboptimal separation between loss function and hash function. In MAC, both elements are decoupled within each iteration, while still optimizing the correct objective: the step over the hash function does not involve the loss, and the step over the codes does not involve the hash function, but both are iterated. The connection between both steps occurs through the auxiliary coordinates, which are the binary codes themselves. The penalty regularizes the loss so that its optimal codes are progressively closer to what a hash function from the given class (e.g. linear) can achieve.

What is the best type of hash function to use? The answer to this is not unique, as it depends on application-specific factors: quality of the codes produced (to retrieve the correct images), time to compute the codes on high-dimensional data (since, after all, the reason to use binary hashing is to speed up retrieval), ease of implementation within a given hardware architecture and software libraries, etc. Our framework facilitates considerably this choice, because training (optimally) different types of hash functions simply involves reusing an existing classification algorithm within the $\mathbf{h}$ step, with no changes to the $\mathbf{Z}$ step.

In terms of runtime, the resulting MAC algorithm is not much slower than the suboptimal two-step approach: it is comparable to iterating the latter a few times. Besides, since all iterations except the first are warm-started, the average cost of one iteration is lower than for the two-step approach.

The method of auxiliary coordinates has also been applied in the context of binary hashing to a different objective function, the binary autoencoder (BA) (Carreira-Perpiñán and Raziperchikolaei, 2015):

$$E_{\mathrm{BA}}(\mathbf{h},\mathbf{f}) = \sum_{n=1}^{N} \left\| \mathbf{x}_n - \mathbf{f}(\mathbf{h}(\mathbf{x}_n)) \right\|^2 \tag{13}$$

where $\mathbf{h}$ is the hash function, or encoder, and $\mathbf{f}$ is a decoder. As with the pairwise loss function, the MAC algorithm alternates between fitting the hash function (and the decoder) given the codes, and optimizing over the codes. However, in the binary autoencoder the optimization over the codes decouples over every data point (since the objective function involves one term per data point). This has an important computational advantage in the $\mathbf{Z}$ step: rather than having to solve one large optimization problem $\{\mathbf{z}_1,\dots,\mathbf{z}_N\}$ over Nb binary variables, it has to solve N small optimization problems $\{\mathbf{z}_1\},\dots,\{\mathbf{z}_N\}$ each over b variables, which is much faster and easier to solve (since b is relatively small in practice). Also, the BA objective function does not require any neighborhood information (e.g. the affinity between pairs of neighbors) and scales linearly with the dataset. Computing these affinity values, or even finding pairs of neighbors, is computationally very costly. For these reasons, the BA can scale to training on larger datasets than affinity-based loss functions. The BA objective function does have the disadvantage of being less directly related to the goals that are desirable from an information retrieval point of view, such as precision and recall.

7 Conclusion

We have proposed a general framework for binary hashing using affinity-based loss functions. It improves over previous, suboptimal approaches based on learning binary codes first and then learning the hash function. Instead, it optimizes over the binary codes and the hash function in alternation, ensuring that the final codes and hash function jointly optimize the loss function. This was possible by introducing auxiliary variables that conditionally decouple the codes from the hash function, and gradually enforcing the corresponding constraints. Our framework makes it easy to design an optimization algorithm for a new choice of loss function or hash function: one simply reuses existing software that optimizes each in isolation. The resulting algorithm is not much slower than the suboptimal two-step approach: it is comparable to iterating the latter a few times.

The step over the hash function is essentially a solved problem if using a classifier, since this can be learned in an accurate and scalable way using machine learning techniques. The most difficult and time-consuming part in our framework is the optimization over the binary codes, which is NP-complete and involves a large number of binary variables and of terms in the objective function. Although some techniques exist (Lin et al., 2013, 2014) that produce practical results, finding algorithms that reliably find good local optima and scale to training on large datasets is an important topic of future research.

Another direction for future work involves learning more sophisticated hash functions that go beyond mapping image features onto output binary codes using simple classifiers such as SVMs. This is possible because the optimization over the hash function parameters is confined to the **h** step and takes the form of a supervised classification problem, so we can apply an array of techniques from machine learning and computer vision. For example, it may be possible to learn image features that work better with hashing than standard features such as SIFT, or to learn transformations of the input to which the binary codes should be invariant, such as translation, rotation or alignment.

Acknowledgments

Work supported by NSF award IIS–1423515.

References

A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. ACM*, 51(1):117–122, Jan. 2008.

M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, June 2003.

Y. Boykov and V. Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *Proc. 9th Int. Conf. Computer Vision (ICCV'03)*, pages 26–33, Nice, France, Oct. 14–17 2003.

Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, Sept. 2004.

M. Á. Carreira-Perpiñán. The elastic embedding algorithm for dimensionality reduction. In J. Fürnkranz and T. Joachims, editors, *Proc. of the 27th Int. Conf. Machine Learning (ICML 2010)*, pages 167–174, Haifa, Israel, June 21–25 2010.

M. Á. Carreira-Perpiñán and R. Raziperchikolaei. Hashing with binary autoencoders. arXiv:1501.00756 [cs.LG], Jan. 5 2015.

M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. arXiv:1212.5921 [cs.LG], Dec. 24 2012.

M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. In S. Kaski and J. Corander, editors, *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pages 10–19, Reykjavik, Iceland, Apr. 22–25 2014.

T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y.-T. Zheng. NUS-WIDE: A real-world web image database from National University of Singapore. In *Proc. ACM Conf. Image and Video Retrieval (CIVR'09)*, Santorini, Greece, July 8–10 2009.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *J. Machine Learning Research*, 9:1871–1874, Aug. 2008.

Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A Procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 35(12):2916–2929, Dec. 2013.

J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In *Proc. of the 2012 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'12)*, pages 2957–2964, Providence, RI, June 16–21 2012.

J. J. Hull. A database for handwritten text recognition research. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 16(5):550–554, May 1994.

H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan. 2011.

R. Kohavi and G. H. John. The wrapper approach. In H. Liu and H. Motoda, editors, *Feature Extraction, Construction and Selection. A Data Mining Perspective*. Springer-Verlag, 1998.

V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(2):147–159, Feb. 2003.

A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Dept. of Computer Science, University of Toronto, Apr. 8 2009.

B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 22, pages 1042–1050. MIT Press, Cambridge, MA, 2009.

B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(6):1092–1104, June 2012.

G. Lin, C. Shen, D. Suter, and A. van den Hengel. A general two-step approach to learning-based hashing. In *Proc. 14th Int. Conf. Computer Vision (ICCV'13)*, pages 2552–2559, Sydney, Australia, Dec. 1–8 2013.

G. Lin, C. Shen, Q. Shi, A. van den Hengel, and D. Suter. Fast supervised hashing with decision trees for high-dimensional data. In *Proc. of the 2014 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'14)*, pages 1971–1978, Columbus, OH, June 23–28 2014.

W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In L. Getoor and T. Scheffer, editors, *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, pages 1–8, Bellevue, WA, June 28 – July 2 2011.

W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. In *Proc. of the 2012 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'12)*, pages 2074–2081, Providence, RI, June 16–21 2012.

D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Computer Vision*, 60(2): 91–110, Nov. 2004.

S. A. Nene, S. K. Nayar, and H. Murase. Columbia object image library (COIL-20). Technical Report CUCS–005–96, Dept. of Computer Science, Columbia University, Feb. 1996.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

M. Norouzi and D. Fleet. Minimal loss hashing for compact binary codes. In L. Getoor and T. Scheffer, editors, *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, Bellevue, WA, June 28 – July 2 2011.

A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Computer Vision*, 42(3):145–175, May 2001.

S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290 (5500):2323–2326, Dec. 22 2000.

C. Strecha, A. M. Bronstein, M. M. Bronstein, and P. Fua. LDAHash: Improved matching with smaller descriptors. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(1):66–78, Jan. 2012.

L. J. P. van der Maaten. Barnes-Hut-SNE. In *Int. Conf. Learning Representations (ICLR 2013)*, Scottsdale, AZ, May 2–4 2013.

L. J. P. van der Maaten and G. E. Hinton. Visualizing data using *t*-SNE. *J. Machine Learning Research*, 9: 2579–2605, Nov. 2008.

M. Vladymyrov and M. Á. Carreira-Perpiñán. Partial-Hessian strategies for fast learning of nonlinear embeddings. In J. Langford and J. Pineau, editors, *Proc. of the 29th Int. Conf. Machine Learning (ICML 2012)*, pages 345–352, Edinburgh, Scotland, June 26 – July 1 2012.

M. Vladymyrov and M. Á. Carreira-Perpiñán. Linear-time training of nonlinear low-dimensional embeddings. In S. Kaski and J. Corander, editors, *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pages 968–977, Reykjavik, Iceland, Apr. 22–25 2014.

J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for large scale search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(12):2393–2406, Dec. 2012.

Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In D. Koller, Y. Bengio, D. Schuurmans, L. Bottou, and A. Culotta, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 21, pages 1753–1760. MIT Press, Cambridge, MA, 2009.

Z. Yang, J. Peltonen, and S. Kaski. Scalable optimization for neighbor embedding for visualization. In S. Dasgupta and D. McAllester, editors, *Proc. of the 30th Int. Conf. Machine Learning (ICML 2013)*, pages 127–135, Atlanta, GA, June 16–21 2013.

S. X. Yu and J. Shi. Multiclass spectral clustering. In *Proc. 9th Int. Conf. Computer Vision (ICCV'03)*, pages 313–319, Nice, France, Oct. 14–17 2003.

D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. In *Proc. of the 33rd ACM Conf. Research and Development in Information Retrieval (SIGIR 2010)*, pages 18–25, Geneva, Switzerland, July 19–23 2010.

C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: FORTRAN subroutines for large-scale bound-constrained optimization. *ACM Trans. Mathematical Software*, 23(4):550–560, Dec. 1997.